Abstract—Mobile app testing and evaluation requires exposing the app to a wide array of real world context conditions viz. location, sensor values, network conditions etc. Such comprehensive context conditions are difficult to create in a development environment on a real device, therefore, simulating them in a mobile emulator is a promising alternative. We present ContextMonkey, a framework for context generation in a mobile emulator. ContextMonkey can generate realistic context by leveraging traces in correlated and interdependent way from heterogeneous sources: remote trace databases, models, and trace files. It eases the burden on developers to collect correlated traces, convert them to a common format, and feed them to the emulator in an orchestrated manner. We present examples demonstrating the simplicity and potential of ContextMonkey in app testing scenarios.

I. INTRODUCTION

The growing popularity of mobile devices like smart-phones, tablets, and wearables has created a highly-competitive mobile app ecosystem where quality of the user experience is crucial for the success of the app. Recent studies [1], [2] have found that 44% users delete their apps immediately if they do not perform as per their expectations. Bad experiences are attributed to heterogeneous factors such as ease of user interfaces as well as poor app performance over networks and across different mobile devices etc. Thus, pre-release app testing and evaluation is critical.

The app performance in terms of crashes, response times, energy consumption, network bandwidth consumption, etc. is significantly impacted by the context created by the complex interaction of multiple factors related to the execution environment. These include mobile device capabilities, the quality of the network connectivity to the backend, the load on the backend servers, the type and frequency of sensor values, and users location among others. Furthermore, these factors are often interrelated, and dependent on each other. The app needs to be exposed to a wide array of realistic combinations of all these parameters in the test phase so as to reduce performance issues post release. Recreating such rich context in the physical reality introduces high overheads in time and cost, and challenges in controllability and repeatability. Thus, realistic trace-driven simulation of such context conditions can be a viable alternative.

However, creating trace-driven context simulation is quite laborious. One would have to create and build a library of wide variety of realistic traces and play them in an orchestrated manner. This is not straightforward since the traces for the individual context modalities might have to be collected in a correlated manner and from heterogeneous sources. They can be recorded using mobile devices, leveraged from trace databases, generated using mathematical models etc. All the traces would then have to be converted to a common format. Further, these simulated conditions would need to be made available to the app under test. There are two possible approaches to do this: using either a real device, or a mobile emulator. Linking such a simulation to a real device is operating system (OS) platform dependent, and not transparent. It would require developers to include the simulation as a library in the app source code, and modify it to use the simulation APIs instead of the standard APIs provided by the OS. On the other hand, mobile emulators imitate internal hardware of mobile devices in a software stack, and have the functionality that enables the software stack to communicate with the app without having to make any changes to the app source code. They also have existing APIs exposed to configure the stack appropriately. These APIs can be reused to simplify the linking of a context simulation by feeding traces in an orchestrated manner. Despite being widely used, mobile emulators lack comprehensive capability for context simulation, which is increasingly becoming important with sensor-rich apps. Thus, linking context simulations to emulators can help to bridge the gap between them and real devices, increasing their level of realism.

In this paper, we present ContextMonkey, a framework that simulates realistic contextual conditions for app testing and evaluation in a mobile emulator. ContextMonkey enables context simulation by leveraging traces from external trace databases, models, libraries, and trace files in a correlated and interdependent manner. It automates the fetching of traces from these heterogeneous sources, and converts them into emulator specific format, and then orchestrates their feeding to the emulator. Furthermore, it provides a set of APIs using which the developers can define suitable contextual simulations as per the requirements of the app under test. ContextMonkey framework is emulator independent, and can be used for simulating context in multitudes of mobile emulators running on the local workstation as well as those running in the cloud based services.

II. CONTEXT MODELING FOR APP TESTING & EVALUATION

Many definitions of Context have been proposed in the literature (see [3]). Dey [4] defines context as “any information
that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves”. Context for mobile apps can be understood as a set of interrelated factors that can potentially impact the end behavior of the app. We refer to these factors as modalities further on in this paper. These modalities include: 1) Mobile device capabilities - CPU, memory, camera, storage. 2) Sensors - temperature, pressure, humidity, light intensity, noise sound level, and change in their values. 3) User state, location, and environment: stationary, mobile, indoors/outdoors, etc. 4) Network type and conditions - 3G, LTE, WiFi - Latency, bandwidth, and packet loss.

These modalities are low-level, often interdependent on each other, and have values that may vary over time. The context logic in mobile apps consumes these low-level modalities, aggregates them, and uses them for context estimation, adapting their behavior over time accordingly. Thus, firstly, ContextMonkey needs a context model which is based on low-level modalities that vary over time, as this would enable the context logic in the apps to function seamlessly. Secondly, the model needs to provide support for defining inter-dependencies between the low-level modalities as observed in the real-world. And finally, the model should be easy to learn and use by developers and at the same time be able to express wide array of high-level context definitions. This is due to the fact that in an app testing and evaluation, the appropriate high-level context definition to be used, the relevant modalities associated with them, and the dependencies among the modalities are largely app under test specific, and only best known to the developers.

Context Modeling has been well studied in the literature (see [5], [6], [7]). There are several approaches to model the context. Key-value model is the simplest way to represent context information as key-value pairs. However, it cannot model the inter-dependencies and relationships between modalities. Markup based models (ContextML [8], Composite Capabilities/Preference Profiles (CC/PP) [9], etc.) represent context data as tags that enable efficient retrieval and validation. Markup based models are highly app dependent and increase in complexity as the information at multiple levels increase, hence, they are not preferable. Logical models use rules, expressions, and facts to represent context. They have high-level of expressibility, however are strongly coupled to the application. Ontology based models [10] are the most popular choice due to its semantic reasoning, application independence, and expressiveness capabilities. However, representation of Ontology based models can be complex, and they are generally resource intensive. Graph based models [11], [12] use relationships to represent context. Such a model offers a middle ground in terms of variation in representation of low-level relationships, richness of expression, and ease of use due to its intuitive nature. Therefore, for ContextMonkey we have decided to use graph based models.

In the following we give an example illustrating its use, by modeling a simple context scenario A Walk at the Brandenburg Gate using low-level modalities. Such a context might be defined by the facts that 1) The mobile device should have the location with latitude and longitude values of the Brandenburg Gate. 2) The device should experience similar network conditions as at the Brandenburg Gate for a given network carrier. 3) The camera on the mobile device should capture image of the Brandenburg Gate when pointing towards it, and 4) the accelerometer on the mobile device should have values as in real walking. The context is represented as a Directed Acyclic Graph (DAG) \( G = \langle V, E \rangle \) (shown in Figure 1), where \( V \) is the vertices set representing all the modalities in the context viz. location, camera, compass, network, and accelerometer. \( E \) is the edge set, where each edge \( e \) represents the interdependence between two modalities and implies that the modality \( V_k \) is dependent on \( V_j \). In our example, the camera modality is dependent on both the location and the compass, and the network modality is only dependent on the location. The acyclic nature of the graph prevents circular dependencies between the modalities. The dependencies in the context graph are transitive by default. The context graph can have disconnected components that represent modalities independent from others, for example, accelerometer modality. The temporal variation in the modality is specified using the rate of change parameter. Each modality has its own rate of change.

III. CONTEXTMONKEY

A. Design Goals

The ContextMonkey framework design is based on achieving three important goals that are discussed in the following.

1) Emulator and Data Source Independence: The framework should be decoupled from both the emulator as well as the data sources used in obtaining the traces for simulation. This ensures seamless integration of the framework with various emulators (those offering suitable interface to feed the traces) available for smart-phones, tablets, wearable devices, and heterogeneous data sources for traces (those offering interface to obtain traces) such as crowd-sourced databases, models, and trace files.

2) Extensibility: The framework should promote easier integration of new high-level context scenarios in the future. This implies that it should be possible to add new low-level modalities to the framework. These low-level modalities can be new physical and virtual sensors (functions over physical sensor values) added on the emulators due to growing mobile device capabilities, or any modality that exist on current mobile device but not supported by its emulator presently.

3) Fidelity & Seamlessness: The framework should simulate context for the app in an emulator-agnostic manner, just as on
a real device. It should capture context at fine-grained level in high-fidelity manner; feeding traces of each individual modality to the emulator at rates observed on a real mobile device. It should also adapt the trace feeding rates of modality as per the rates configured in the app source code. Furthermore, it should provide context simulation to the app seamlessly, without requiring to make any changes in the app source code like integrating external libraries, and API change.

B. Challenges
One of the key challenges in the context simulation with regards to the above design considerations is the potential interdependent nature of the modalities. Interdependence requires careful orchestration in fetching and feeding traces to the emulator. This is not easy when the data source types of modalities differ. Each data source may have its own method and APIs to generate traces; the time for fetching traces from each data source type may differ widely. Further, the data source may have its own format used to represent the trace. On the other hand, the emulator would have its own APIs for feeding traces, and trace format understood by the emulator might differ from that of the data source. Therefore, the framework needs to understand and execute the respective API calls for the heterogeneous data sources and emulators, and also handle data format conversions. These tasks have to be executed respecting the defined rate of change of the individual modalities so as to maintain high fidelity.

C. Architecture
The ContextMonkey architecture has been proposed in the poster [13]. The architecture is component based and decoupled into three layers (shown in Figure 2).

1) Trace Layer: The Trace Layer is primarily responsible for interacting with heterogeneous trace data sources used in simulation, and orchestrating fetching of traces from them. It primarily supports three data source types:

- **Trace database** services in the cloud such as Google StreetView [14], Open Signal [15], Foursquare location [16], etc. that support fetching traces using API queries.
- **Models** such as BonnMotion [17], SUMO [18], etc. that provide highly specialized tools & libraries to generate traces.
- **Trace files** that contain individual modality trace datasets captured using real mobile devices (such as those available at Crawdad [19]).

The Trace Layer includes three components:

**Data Source Manager** - The Data Source Manager coordinates all the Trace Layer functionalities associated with obtaining traces from the data sources and processing them for the simulation. It constructs trace fetching requests. In case of trace database services this includes API URLs with appropriate query parameters, authorization and authentication necessary for using APIs, the HTTP request method, etc. In case of models, it is the library path details, the appropriate commands and their corresponding options needed to generate traces, and file paths, headers and their respective formats for trace files. It also handles the asynchronous Trace API methods that are explained further on. Using the data source definitions, the manager maps incoming trace fetching requests to the appropriate data source handler. The manager also handles processing of traces fetched from a data source. The data source trace value units for the modality could be different and need to be converted. For instance, in case of camera modality images might have different resolution, file compression format which need to be processed. The manager filters unwanted attributes from the fetched traces. It also handles trace fetching failures that can occur due to heterogeneous reasons: unavailability of traces, exceeded limits on the API calls, service unavailability, network failures, etc. In case of a failure, the manager supports trace generation based on default values, interpolation, and extrapolation functions. Finally, it sends processed traces to the Trace Format Manager and forwards format converted traces received from the Trace Format Manager to the Context Layer.

**Trace Format Manager** - A data source might have its own trace representation format which could be different from the emulator understood trace format. The Trace Layer is emulator-agnostic, therefore the Trace Format Manager facilitates conversion from the data source specific trace formats to a common internal representation format of ContextMonkey.
**ContextMonkey External API - To the app developers**

- createContextSimulation - create an instance of ContextMonkey simulation environment
- createContextGraph - create an instance of Context Graph to be used in the simulation
- addModality - include a new modality or modalities in the Context Graph
- addDependency - add a modality dependency or dependencies in the Context Graph
- addDataSource - associate a data source with the modality
- runTime - specify the context simulation run time period in seconds
- start - start the context simulation

**Fig. 3: ContextMonkey API Description**

**Data Source Handler** - The Data Source Handler contains the connection logic for fetching traces from the data sources. It multiplexes execution of trace fetching requests from various data sources. It executes the queries for fetching trace values from the database, commands for models, and read operation on trace files and forwards them to the Data source manager. It maintains a local cache of fetched traces, and only fetches a trace in case of a cache miss.

2) **Context Layer** - The Context Layer lies at the heart of ContextMonkey managing the control and the execution of the simulation. It acts as a bridge between fetching traces from the data sources, and feeding them to the emulator. It is both data source, as well as mobile emulator-agnostic.

**Context Data Manager** - The Context Data Manager handles all the functionality of the Context Layer. It maintains the context model definition that includes the context graph, the physical, and the virtual modalities, and its dependencies. It computes virtual modality values using these definitions. The manager maintains an individual real clock timer for each modality which periodically fires according to its rate of change parameter, and calls the asynchronous Trace API in order to request traces. In order to obtain a modality trace that is dependent on another, the manager keeps the knowledge of the current value of its predecessor modalities in the context graph. Furthermore, it registers, and implements the callback API function to be notified of successful trace requests. Upon receiving the requested trace, the registered callback function forwards the trace to the Feed Layer.

**Context Data Queue** - ContextMonkey also supports defining virtual modalities. The virtual modality computation requires trace values of dependent physical modalities. As the physical modalities can have varying amount of time delay in fetching their trace, the queue stores a copy of their fetched trace value before forwarding it to the Feed Layer. Each modality has its own separate queue. Separate queues ensure sequential control logic for individual context modality traces. Computing virtual sensor function requires a synchronous dequeue from dependent modality queues.

3) **Feed Layer** - The Feed Layer provides the glue which is necessary to bind ContextMonkey to a compatible mobile emulator. Compatibility implies that a mobile emulator should support a suitable API for feeding traces, and also make them available to the app. The main components of Feed Layer are:

- **Emulator Manager** - The Emulator Manager is the core component that coordinates activities of the Feed Layer. It maintains information necessary for integrating mobile emulator with ContextMonkey. This includes emulator specific information about different supported modalities by the emulator, the emulator API description of a modality for setting and getting values, their respective formats, the communication protocol and configuration parameters to be used in calling the emulator API. Furthermore, it also handles the asynchronous Feed API exposed to the Context Layer to set and get the individual modality values from the emulator. It forwards received traces to the Feed Format Manager for trace format conversion. Finally, it maps the modality trace set and get requests to the appropriate handler for feeding the trace.

- **Feed Format Manager** - The Feed Format Manager deals with the trace format conversions between the ContextMonkey’s internal representation format, and the emulator data format.

- **Emulator Handler** - The Emulator Handler is responsible for establishing connection to the emulator using the emulator specified protocol. It executes requests to set and get the modality values in the emulator by making appropriate API calls.

4) **API Description** - The ContextMonkey API description is given in Figure 3. The external APIs are those that are exposed to the app developers to enable them to specify a context scenario for simulation in the form of graphs, and configure the simulation related parameters. This includes defining modalities and their dependencies, the rate of change, registering data sources and their APIs, the emulator and its APIs, the trace formats, and simulation run period. The internal Trace API and the Feed API are used for the internal communication between the three layers for fetching the traces and feeding them to the emulator.

**IV. DESIGN ALTERNATIVES**

During ContextMonkey design, we considered several alternatives, some were not obvious at the time, however quite important in refining the design goals. We discuss them in the following.

**A. System Model Alternatives**

There are two possible models to implement the ContextMonkey architecture. In a local model, the framework runs on the developer’s local workstation that has an app development environment setup alongside a mobile emulator. For instance, Android SDKs [20] provide toolchains, platform system images, external libraries, and APIs along with emulators for app development and testing. Alternatively, developers may test their apps remotely on a mobile emulator that is running on a Virtual Machine (VM) in the cloud. CrowdMeter [21], Caipa [22] are some platforms that offer such cloud based testing services. The cloud system model, due to its scalability, offers an additional degree of flexibility by loosely coupling the ContextMonkey layers among multiple VMs. We limit the implementation of the architectural layers on to a single dedicated machine. However, we consider both the local workstation and the cloud VM for the evaluation.
B. Layer Organization

One of the most simple ways to realize the ContextMonkey architecture is to place the ContextMonkey as a mobile app inside the guest mobile OS in the emulator. Besides losing the emulator and OS platform independence, and seamlessness, it introduces overhead from having to run the app in an emulated environment where code translation from the guest to the host system is necessary. The other four alternatives (illustrated in Figure 4) stem from the possibilities to structure the layers between the emulator and the underlying host platform. The first alternative places the three layers within the mobile emulator. In case of the second, and in the third alternative, the layers are distributed between the emulator and the host platform. However, these three alternatives fail to achieve complete emulator independence. The fourth alternative is comprised of placing all the layers on the host platform external to the emulator, and reflects our design goals.

C. Data Trace Representation

Feeding a trace to an emulator requires format conversion from a data source format to an intermediate common format, and then to an emulator understood format. XML, JSON, and Protobuf [23] are some of the data formats that can be used for representing traces. Both XML and JSON are human-readable, and offer encoding and decoding without the knowledge of the schema in advance. However, compared to XML, JSON is compact, and less verbose. A binary format like Protobuf needs a predefined schema that needs to be compiled before using it for encoding and decoding. Being a binary format, Protobuf can seamlessly support encoding and decoding of binary trace modalities like images. Furthermore, it has a smaller footprint compared to XML, JSON, and also offers faster data encoding and decoding. The trace feeding rate is an important metric in order to achieve high fidelity for context simulation. Therefore, we use Protobuf for representing the traces internally within the ContextMonkey.

V. IMPLEMENTATION

The ContextMonkey prototype has been implemented in Python. Each architectural layer is organized as a separate Python module, and the components are implemented as separate classes within the module. The individual components in the implementation are shown in Figure 5. At the heart of ContextMonkey implementation lies the Twisted [24] library that provides support for event driven network programming. ContextMonkey is available as open-source project at https://github.com/manojrege/contextmonkey. The prototype consists of ≈ 8K lines of Python code. In the following, we discuss the implementation in detail.

The contextmodel module includes class definitions for the modality and the Context Graph (DAG) used in expressing context within the framework. The ContextModality abstract class defines an individual context modality and its generic attributes like rate of change, modality dependencies, data-source, and trace-processing library. PhysicalContextModality and VirtualContextModality are two subclasses that extend the ContextModality class. The PhysicalContextModality class represents sensors, location sensors, camera, and network condition modalities within the context. The VirtualContextModality class represents virtual sensors and has computation attribute specifying the method used to calculate the sensor value. The data-source associated with a context modality are defined in the DataSource abstract class and includes attributes such as datasourcetype, datasourceformat, queryparameterlist associated with the data-source. The Database, the Model, and the TraceFile classes extend the DataSource class and represent the three types of sources supported by ContextMonkey. The ContextGraph class represents the DAG for context specification. The DAG is implemented using the python NetworkX library. The node attribute in the DAG are of type PhysicalContextModality and VirtualContextModality, and the edge attributes define the dependencies among them. It also provides methods for getting and adding context modalities in the graph, and performs cyclic dependency checks in the context model.

The ContextSimulation class exposes ContextMonkey APIs to the developers. It maintains simulation definition, environment, and the associated global variables. The ContextEngine class includes the event based Twisted reactor loop that coordinates and schedules the events in each layer. It is also responsible for starting and stopping the simulation. The developers need to import the contextmodel and the ContextSimulation class to create and run a context simulation.

The ContextManager class is responsible for generation of physical modality trace requests based on the modality rate of change. It includes a list of timers for each modality based on the TimerService utility from the Twisted library. When the modality timer fires a trace request is sent by calling the getValue method exposed by the Trace Layer.

The DataSourceManager class handles all the Trace Layer functionality. Its executeFetch method dynamically instantiates and loads a DataSourceHandler class instance for trace fetching. This handler instance then dynamically maps the request to either one of the DatabaseRequestHandler, or the FileRequestHandler, or the ModelRequestHandler class instances, based on its modality type. The DatabaseRequestHandler class is extendable and
provides a set of methods for fetching traces from remote databases. Presently, it supports fetching using HTTP and HTTPS based GET method. It uses the Twisted WebClient that enables sending specific request methods, setting appropriate headers and also supports error handling based on response codes. Similarly, the FileRequestHandler, and the ModelRequestHandler classes support trace fetching methods from the files and the models respectively. Some models like BonnMotion provide commands to generate traces and store them in a file. The ModelRequestHandler then delegates the trace fetching to FileRequestHandler. The trace fetching requests are handled asynchronously by the underlying Twisted reactor loop.

The TraceProcessing class provides a generic interface that provides methods to filter and process trace for each context modality. As these are data source dependent, the developer can use a separate class to implement these methods and enable customized trace processing. Presently, it supports processing of camera and network modality traces from StreetView and OpenSignal databases respectively. The support for trace format conversion from data source format to Protobuf is implemented in the TraceFormatManager class. The current version supports conversion between the comma separate value (CSV), JSON and byte formats, and Protobuf.

The ContextDataQueue class provides support for trace buffering in order to be utilized for the computation of virtual sensor value. The traces from the buffer are dequeued as per the feed rate. The EmulatorManager class exposes a setModalityValue method for this purpose. The FeedFormatManager class implements format conversion from Protobuf to emulator understood format. The EmulatorHandler provides a generic Connection interface that contains method to establish a connection with the emulator and set the trace values of individual modality in the emulator using the emulator provided API. We have implemented connection interfaces for the stock Android [25] (AndroidConnection) and Genymotion [26] (GenymotionConnection) emulators.

The control flow in each layer is implemented using the Deferred design pattern provided by the Twisted framework. It registers a sequential callback chain on the deferred objects in each stage, consisting of trace request generation, datasource handler mapping, trace request fetching, trace processing, format conversion, and trace feeding.

VI. Evaluation

A. Emulator Independence

In order to evaluate the emulator independence, we present examples demonstrating ease of integration of ContextMonkey with stock Android [25], and Genymotion [26] emulators.

1) Stock Android: The stock Android emulator comes bundled with Android SDK Platform [20], and uses the Quick EMUlator (QEMU) [27] virtualization suite that provides emulation support for several different CPU architectures. The emulator runs Android guest operating system with the Goldfish kernel in QEMU. The emulator supports sensor simulation stack with a Telnet based command interface.
The command interface enables setting trace values in the Android Hardware Abstraction Layer (HAL) that are then made available to the app space in the emulator. We successfully integrated the stock Android Emulator with our ContextMonkey framework. The integration module consists of 338 lines of code. The module contains the AndroidConnection class that implements the interface exposed by the Emulation Handler for connection establishment and API call logic. It establishes a Telnet session over a persistent TCP socket connection. Each modality is implemented in a separate class that extends from GeneralSensorProtocol class, and provides method for setting trace values. We implemented support for 7 sensor modalities with the stock Android emulator.

### TABLE I: Details of emulator integration with ContextMonkey

<table>
<thead>
<tr>
<th>Emulator</th>
<th>Stock Android</th>
<th>Genymotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>338</td>
<td>399</td>
</tr>
<tr>
<td>Number of sensors</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

2) **Genymotion**: Genymotion provides a custom GenyShell shell environment to communicate with the emulator. Similar to the stock Android emulator, the connection and the API call logic is implemented through Emulator Handler provided interface in GenymotionConnection. The integration source code for Genymotion emulator consists of 399 lines of code. The connection spawns a Telnet server process and the GenyShell as a separate daemon process. The inter process communication between the Telnet server and the GenyShell process is carried using the ProcessProtocol provided by Twisted. The API calls to set the trace modality are sent as commands to the Telnet server that forwards them to GenyShell to set the values in the emulator.

### B. Data Source Independence

To demonstrate the data source independence, we integrated modalities from popular external trace databases such as StreetView [14], OpenSignal [15], and FourSquare [16] with ContextMonkey (shown in Table II). We also implemented support for location, mobility modality trace generation using popular models like BonnMotion [17], and libraries like SUMO [18]. Furthermore, modality traces from datasets available on CRAWDAD [19], like taxi mobility traces [28], and accelerometer sampled traces collected from Android smartphones while driving on different vehicles [29] were also added.

### TABLE II: Description of different data sources integrated with ContextMonkey

<table>
<thead>
<tr>
<th>Modality</th>
<th>Source</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>FourSquare</td>
<td>Database</td>
</tr>
<tr>
<td></td>
<td>SUMO</td>
<td>Model</td>
</tr>
<tr>
<td>Mobility</td>
<td>BonnMotion</td>
<td>File</td>
</tr>
<tr>
<td>Camera</td>
<td>Google StreetView</td>
<td>File</td>
</tr>
<tr>
<td>Network</td>
<td>OpenSignal</td>
<td>Database</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>Mobile Device (driving)</td>
<td>File</td>
</tr>
</tbody>
</table>

We demonstrate the ContextMonkey extensibility using a MobiCamSim example. MobiCamSim is a trace-driven location context based camera simulation that can be used for testing camera sensing based mobile apps. The present camera modality in the stock Android emulator serves either the checker-board or the webcam images. We introduce a new camera modality in the stock Android emulator and implement the support for feeding image traces to the camera through an API. We then use ContextMonkey to define a new context simulation that includes location, and compass dependent camera modality with a predefined field of view parameter. We use the StreetView database to feed location based image traces to MobiCamSim (shown in Figure 6).

### C. Extensibility: MobiCamSim - A location context driven mobile camera

We demonstrate the ContextMonkey extensibility using a MobiCamSim example. MobiCamSim is a trace-driven location context based camera simulation that can be used for testing camera sensing based mobile apps. The present camera modality in the stock Android emulator serves either the checker-board or the webcam images. We introduce a new camera modality in the stock Android emulator and implement the support for feeding image traces to the camera through an API. We then use ContextMonkey to define a new context simulation that includes location, and compass dependent camera modality with a predefined field of view parameter. We use the StreetView database to feed location based image traces to MobiCamSim (shown in Figure 6).

### D. Fidelity

We measure fidelity by comparing the modality vector reporting rate in an app running on a real mobile device to ContextMonkey instrumented stock Android emulator. The selected modalities include sensors, location, and camera. We use only stock Android emulator for our evaluation as the Genymotion emulator has a limited simulated sensor stack API support. For our experiments, we use a Moto G phone running Android Lollipop 5.1.1. The stock Android emulator has an equivalent hardware with respect to ARM CPU, memory, sensors, and OS configuration. We report results from 10 trial runs. We measure the ContextMonkey simulation fidelity on a local work station (2.3 GHz Intel Core i5 processor with 4GB RAM and Ethernet connectivity), and in our CrowdMeter [30] cloud testbed (t2.medium virtual machine with 2 virtual CPU cores, and 4.0 GB RAM from Amazon Web Services (AWS) Elastic Compute Cloud (EC2)).

**Sensors** - We use accelerometer, and compass modality traces captured on a Moto G phone in a file to measure the sensor simulation fidelity. For the sake of brevity, we only present results for the accelerometer modality. Our analysis is as well applicable to other sensors such as temperature, rotation, gravity, gyroscope, light, etc. present on typical Android devices. In the app, we configure four delay settings provided in the Android API for reporting sensor vector value events: FASTEST, GAME, UI, NORMAL, and measure their respective average reporting rates over a 2 min interval. FASTEST, GAME, UI, NORMAL settings report events at 100 Hz, 50 Hz, 20 Hz, and 5 Hz respectively. We set these values as feeding rates in

![Image](362x655 to 513x741)

**Fig. 6:** MobiCamSim: A location based mobile camera simulation with the StreetView in the stock Android emulator.
the ContextMonkey based accelerometer simulation, and then measure the reporting rate in the app within the emulator. We observed that the maximum achievable average reporting rate in the app in case of simulation is 1.20 Hz (local workstation) and 1.23 Hz (cloud VM). On further investigation, we found that although ContextMonkey is able to feed values to the simulated sensor stack within the emulator at the configured rate, the stack only reports events to the app every 800 ms (1.25 Hz). We modified the emulator source to change the reporting period as per the FASTEST, GAME UI, NORMAL values measured on the Moto G phone. Figure 7(a) shows the bar plots comparing the average event reporting rates (over a 2 min interval) for accelerometer on a Moto G phone, and in a simulation running on the local workstation and a VM in the cloud testbed. Post fine-tuning, the average reporting rate in simulation improved significantly to 47.7 Hz (local workstation) and to 47.9 Hz (cloud VM), but fell short of FASTEST value (100 Hz). Our analysis suggest that the bottleneck lies in the simulated sensor stack that has to report events to the HAL and then to the app space, and the time required for it is larger than 10 ms. Also, the binary opcode translation for the ARM CPU did not play a big role, as only a marginal improvement was observed in the average reporting rate for the emulator with x86 CPU: 48.5 Hz (local workstation) and 48.9 Hz (cloud VM). We plan to address this in our future work. However, ContextMonkey was consistently feeding traces to the emulator at the desired rate for sensor simulation.

**Location** - Location is an important modality for context. We also use trace files generated from a BonnMotion model to measure location simulation fidelity. The Android based devices provide six different types of location data services to the app: cached GPS, cached network (Cell-id, WiFi), realtime GPS, real time network, and passive (piggybacked location updates are received based on other app’s location requests). The Android API also provides minTime and minDistance settings that serve as lower bounds to deliver the location updates to the app. We configure the app to use a GPS provider and obtain location updates as fast as possible (minTime=0 and minDistance=0). We do not consider other factors such as time to get a location fix, number of satellites, accuracy, etc. On a Moto G phone, the GPS provider location update events are reported at the rate of 0.05 Hz. We use this value to configure the location simulation. From the Figure 7(b), one can see that unlike in the sensor simulation, in the location simulation (on both the local workstation, as well as a VM in the cloud testbed), it is possible to achieve a reporting rate that is equivalent to a Moto G phone.

**Camera** - We use MobiCamSim to measure the camera simulation fidelity using capture latency as a metric. The image capture latency is defined as the amount of time taken to capture a picture using a camera app. It also includes the time required to store the picture of a particular resolution on the device file system, or on the internal memory, or the external storage. We use a prototyped Android camera app for the measurements. The resolution of the picture is set to 640 x 640 (maximum resolution provided by StreetView). Figure 7(c) illustrates the bar plots for the camera capture latency measured on a Moto G phone and in MobiCamSim on a local workstation and a cloud VM. The image capture latency of simulation on the local workstation and a cloud VM are 1.14 and 1.04 times that of real Moto G camera.

Overall, we are encouraged by the fidelity results. In future, we plan to address the sensor stack bottleneck issue, through a new stack design for the stock Android emulator.

**E. Microbenchmark - Maximum Achievable Feeding Rate**

We instrumented the ContextMonkey code to measure the time complexity of each individual component in the simulation, and thus, the maximum achievable feeding rate. The maximum achievable feeding rate is measured as the inverse of total time interval starting from the trace request generation to the feeding of trace to the emulator by making an API call. We simulated The walk around Brandenburg Gate context example using the accelerometer, the location, and the location dependent camera and network modalities (details given in Table III).

Table IV shows the component-wise latency breakdown of each modality in the context simulation run on the local workstation and a cloud VM. The maximum achievable feeding rate of a modality trace is largely dependent on its
TABLE IV: ContextMonkey components latency breakdown on the Local Workstation and a Cloud VM. The average, maximum, and minimum statistics are calculated across 100 trial runs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Modality - Data-source</th>
<th>Local Workstation (ms)</th>
<th>Cloud VM (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td>DataSource Handler</td>
<td>Accelerometer - Trace File (JSON)</td>
<td>0.09</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>Location - Model (Text)</td>
<td>0.024</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td>Camera - StreetView (Binary)</td>
<td>97.88</td>
<td>164.88</td>
</tr>
<tr>
<td></td>
<td>Network - OpenSignal (JSON)</td>
<td>867.37</td>
<td>1255.18</td>
</tr>
<tr>
<td>DataSource Manager - Processing</td>
<td>StreetView</td>
<td>42.12</td>
<td>53.11</td>
</tr>
<tr>
<td></td>
<td>OpenSignal</td>
<td>0.09</td>
<td>0.11</td>
</tr>
<tr>
<td>Trace Format Manager - Encoding</td>
<td>Accelerometer - Trace File</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Location - Model</td>
<td>0.14</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>Camera - StreetView</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Network - OpenSignal</td>
<td>0.15</td>
<td>0.19</td>
</tr>
<tr>
<td>Feed Format Manager - Decoding</td>
<td>Accelerometer - Trace File</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Location - Model</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Camera - StreetView</td>
<td>0.09</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>Network - OpenSignal</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>Emulator Handler (API call)</td>
<td>Accelerometer - Trace File</td>
<td>0.42</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td>Location - Model</td>
<td>0.40</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>Camera - StreetView</td>
<td>8.20</td>
<td>9.23</td>
</tr>
<tr>
<td></td>
<td>Network - OpenSignal</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>Total Time</td>
<td>Accelerometer - Trace File</td>
<td>0.49</td>
<td>2.01</td>
</tr>
<tr>
<td></td>
<td>Location - Model</td>
<td>0.50</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td>Camera - StreetView</td>
<td>152.13</td>
<td>221.13</td>
</tr>
<tr>
<td></td>
<td>Network - OpenSignal</td>
<td>867.90</td>
<td>1255.12</td>
</tr>
</tbody>
</table>

data-source type and the trace format. We discuss the least maximum achievable feeding rate observed for different data sources (across 100 trial runs). The least maximum achievable feeding rate for the accelerometer modality using trace files is 497 Hz (local workstation) and 740 Hz (cloud VM). These results support our earlier analysis that the bottleneck lies in the simulated sensor stack of the emulator. In case of the location modality using model, the least maximum achievable feeding rate observed is 489 Hz (local workstation) and 687 Hz (cloud VM). The above results clearly demonstrate that ContextMonkey can easily achieve the feeding rates for high frequency sensors when using trace files, and models. For the camera and the network modality that use a trace database, the maximum achievable feeding rate varies from 4.5 Hz to 8.4 Hz (StreetView) and 0.7 Hz to 1.2 Hz (OpenSignal) on a local workstation. On a cloud VM, the maximum feeding rate increases due to the low end-to-end network path latency and fewer number of hops: 5.5 Hz to 33.2 Hz (StreetView) and 1.05 Hz to 1.35 Hz (OpenSignal). In case of database, the feeding rate is limited by the end-to-end network path latency, and the backend response time (accounting for more than 98%).

F. ContextMonkey Case Studies

We have carried out two case studies using ContextMonkey. Our goal here is to demonstrate the simplicity, and potential of ContextMonkey in the mobile app testing and evaluation, and not the novelty of the apps themselves.

1) GeoCrowdSens: GeoCrowdSens is a crowd-sensing based Android app that uses geofences to trigger collection of data related to surrounding events using mobile devices [32]. The app requests the user to respond to a simple text query; taking picture, video snaps, etc. Geofence defines a geographical area around a point of interest specified by the fine-grained and coarse-grained radius parameters. The coarse-grained radius triggers an injection, and in-app deployment of the geofence over the network, while the fine-grained radius triggers collection of the data when the user enters the area of interest. Reliable triggering of a geofence is dependent on the coarse-grained radius, the user speed, the network conditions, and the location service provider. We use ContextMonkey to simulate context for GeoCrowdSens in the stock Android emulator deployed on a VM in the CrowdMeter testbed. The test set-up includes a hermetic server [33] running locally on the same VM in the testbed. The 3G network is simulated using the in-built network simulation support provided by the emulator (leaky bucket with traffic shaping). For the purpose of simulation, we collected location (GPS location provider) and 3G network traces for different context - walking, running, driving, and commuting (train), using a Moto G phone. The 3G network traces included the end-to-end latency, the bandwidth, and the packet loss between the phone and the GeoCrowdSens backend service. We used Ping utility and Iperf [34] tool for the network trace collection. The location, and the network traces were then included in a file, and used in the ContextMonkey simulation. The simulation consists of 16 lines of code (shown in Appendix A). For the simulation, the rate of change of location is set to 20s. We present the results from 30 trial runs.
for the impact of user speed, and the coarse-grained radius on the detection (hit) percentage of geofences. The hit percentage is the fraction of geofences that were successfully triggered for both the enter and the exit events.

The bar plots in Figure 8 compare the hit percentage in field test and hermetic simulation for the 3G network. As expected, the results show an increasing trend in the hit percentage with the increase in the coarse-grained radius for different mobility scenarios. For walking (5km/h), the coarse-grained radius as small as 20 m can achieve 96.7% hits in hermetic simulation. For average biking speed (20 km/h), 100% hits are achievable for increased radius of 100 m. In case of driving (40-80 km/h), and commuting (train) (>80km/h) 100% hits are obtained for radii values greater than 300 m while radii smaller than 100 m fail to trigger even a single geofence. Next, we compare the hermetic simulation results to a small scale field measurements. The worst case percentage error between field testing and hermetic simulation estimated hit percentages for walking, biking, driving and commuting(train) are 0.03%, 8.3%, 9.8% and 3.7% respectively. These results demonstrate that it is possible to estimate an appropriate threshold coarse-grained radius value based on user speeds and the network using ContextMonkey simulation. We also carried out non-hermetic test simulation experiments, using the location traces from BonnMotion model, and the network statistics that have been reported in the literature [35], [36]. In these experiments, the worst-case percentage error for biking, driving, and commuting(train) increased to 61%, 66%, and 33% respectively while it remained similar in case of walking (0.03%). These test results highlight the impact that the test set-up, and the selected trace data source within ContextMonkey simulation can have on the estimated app performance parameters.

2) CamTour: CamTour is a camera based Android app inspired by Google Glass, and provides information about the landmarks in cities. A user snaps a picture of a landmark; CamTour then uses the location coupled with the picture content to search and deliver landmark related information to the user. Response time is an important performance metric for CamTour. Developers usually measure response times within a limited scope using real devices. While such measurements are quite valuable, obtaining fine-grained Response time analytics based on the real world landmark imagery is desirable for apps such as CamTour. We use ContextMonkey to simulate context for CamTour in the stock Android emulator. We define a context based on a location dependent trace driven camera from StreetView image traces, and the 3G network conditions (latency, bandwidth, and packet loss) from the OpenSignal network traces. The emulator is deployed on a VM in the CrowdMeter testbed. It is not possible to deploy a hermitic setup in the CamTour case study due to an external API dependency [37] for picture content search. Therefore, we use a non-hermitic setup, with the CamTour backend running in the production Google App Engine [38] server. The source code for the ContextMonkey simulation consists of 18 lines of code (given in Appendix B). We measure the CamTour Response time estimates under a given load for 5 different landmarks in 3 different cities, and on for 4 different network carriers.

The results from 20 trial runs are shown in Figure 9. We have labeled the network carriers C1-C4 to mask their identity, and focus on the significance of the results. In Figure 9a, we compare the response times obtained in the simulation with the real measurements obtained from a 3G network carrier in Berlin. The mean response time percentage error between

Fig. 8: A comparison of geofence hit percentage in (hermitic and non-hermitic) simulation with real field for different mobility scenarios.

Fig. 9: Fine-grained CamTour response time analytics on a 3G network using ContextMonkey A comparison of (a) Response time estimates for a network carrier using ContextMonkey on a local workstation and a cloud VM. (b) Response time estimates for different network carriers in Berlin. (c) Response time estimates at different geographical locations.
the simulation and the real measurements is 31.8% on a local workstation and 7.8% in the CrowdMeter cloud testbed. Next, we discuss the response time analytics obtained from the simulations in the CrowdMeter testbed. The app response times across different network carriers in Berlin vary significantly (see Figure 9b). The average 3G latencies on carrier C1 are lower by 73.5% compared to carrier C3. As a result, the average response times observed on C1 are 39.4% lower than C3. Figure 9c illustrates the CamTour estimated response times on a C1 network carrier, across three different cities around the world. It provides insight into the impact of the CamTour users geographical location on the response times.

VII. RELATED WORK

Our work builds on the top of previous research efforts in context modeling, and complements the existing literature in context enabled testing. Context modeling is well studied in the area of Pervasive and Ubiquitous computing. SimuContext [39] is a context simulation framework for testing context aware applications that provides separation between the data sources, and context-aware logic. ContextMonkey, however, is aimed at simulating mobile app specific contexts within mobile emulators. There are a wide array of tools and platforms focusing on different aspects of testing like fault detection, crashes and bugs, UI, functionality etc. For example, AppInsight [40] is a instrumentation framework that focuses on identifying faults within mobile apps. VanarSena [41] utilizes instrumentation to find bugs in the mobile app. Civipa [22] is a scalable cloud based testing service to discover performance bugs and crashes, and hosts a predefined set of contextual trace libraries for black box testing. ContextMonkey nicely complements such platforms and services. It can be utilized by app developers, as well as by large-scale cloud based testing services for context enabled testing. OpenIntents [42] and Samsung Sensor Simulator [43] provide simplified individual sensor simulation capabilities on mobile emulation platforms. The authors in [44] come closest to our goal. They propose a Android based context simulator that uses app source instrumentation with special libraries. In contrast, ContextMonkey does not require app code instrumentation. It offers wide range of possibilities as it provides context simulation support across the emulators using heterogeneous trace data sources.

VIII. CONCLUSIONS

We presented the ContextMonkey framework for context generation in mobile emulators. The key focus of ContextMonkey is to promote use of realistic context in testing and evaluation of mobile apps. Accordingly, the ContextMonkey framework provides cross-emulator support for context generation using trace-driven simulation. It lessens the burden on the developers by automating interdependent trace generation from heterogeneous sources and feeding them to the emulator. Our evaluation of the ContextMonkey framework is based on the performance comparison for two different mobile apps in the real field (on a micro scale) with the simulations generated by the framework in the emulator. In the future, we plan to extend the ContextMonkey framework to support scalable context simulation that can facilitate parallelized and scalable testing of mobile apps.

ACKNOWLEDGMENTS

The work has been partially supported by Amazon AWS Research Grant. A special thanks to our shepherd, Antinisca Di Marco, and the anonymous reviewers, for their suggestions in improving the technical content and presentation of the paper.

REFERENCES

APPENDIX

Context Simulation: GeoCrowdSens

Listing 1: GeoCrowdSens

```python
1 import contextmonkEngine as engine
2 import contextmonkContextModel as cm
3 from contextmonkContextModel import DataSource
4 from contextmonkContextModel import TraceFile
5 from contextmonkContextModel import PhysicalContextModality
6 from contextmonkContextModel import Emulator
7 # Modality and data sources
8 modality = PhysicalContextModality()
9 dataSources = [TraceFile()]
10 # Trace file settings
11 traceFile = TraceFile()
12 # Emulator settings
13 emulator = Emulator()
14 # Start simulation
15 engine.start()
```

B. Context Simulation: CamTour

Listing 2: CamTour

```python
1 import contextmonkEngine as engine
2 import contextmonkContextModel as cm
3 from contextmonkContextModel import DataSource
4 from contextmonkContextModel import TraceFile
5 from contextmonkContextModel import PhysicalContextModality
6 from contextmonkContextModel import Emulator
7 # Modality and data sources
8 modality = PhysicalContextModality()
9 dataSources = [TraceFile()]
10 # Trace file settings
11 traceFile = TraceFile()
12 # Emulator settings
13 emulator = Emulator()
14 # Start simulation
15 engine.start()
```